Задание к курсовой работе по дисциплине

«Теория языков программирования и методы трансляции»

Цель работы: изучение составных частей, основных принципов построения и функционирования трансляторов, практическое освоение методов построения простейших трансляторов для некоторого входного языка, приобретение навыков командной работы.

Задание заключается в создании транслятора с некоторого входного языка на заданный выходной язык. Для выполнения работы студенты делятся на команды по 3 человека. Каждая команда разрабатывает свой входной язык программирования. В качестве выходного (результирующего) должен использоваться язык ассемблера процессоров типа Intel 80x86.

Результатами выполнения курсовой работы являются программная реализация транслятора, пояснительная записка, оформленная в соответствии с требованиями стандартов и задания на работу, и презентация.

Транслятор рекомендуется построить из следующих составных частей:

1. Лексический анализатор.
2. Синтаксический анализатор.
3. Оптимизатор.
4. Генератор результирующего кода.

Для построения транслятора рекомендуется использовать методы, освоенные в ходе выполнения лабораторных работ по курсу «Теория языков программирования и методы трансляции».

Порядок выполнения задания

№	Этап выполнения	Время (недели)	Результат
1.	Получение задания. Формирование команд разработчиков	1	Сформированные команды разработчиков
2.	Описание разработанного языка программирования	3	Учебное руководство по разработанному языку (tutorial, 3 страниц)
		3	Справочник по разработанному языку (reference manual, 5 страниц)
3.	Разработка программы транслятора	8	Работающий транслятор
4.	Защита	1	Окончательный вариант пояснительной записки и презентация
	Итого	16	

Требования к содержанию пояснительной записки

Пояснительная записка должна содержать следующие разделы:

1) Введение (краткое изложение цели работы)
2) Учебное руководство по разработанному языку
3) Справочник по разработанному языку

4) Описание грамматики входного языка в одном из трех возможных видов:
   - форма Бэкуса-Наура;
   - расширенная форма Бэкуса-Наура;
   - графическая форма.
5) Структура разработанного транслятора
6) Описание средств разработки
7) Проблемы, возникшие при разработке транслятора и способы их решения
8) Примеры тестовых программ на входном языке и результирующих программ, сгенерированных транслятором
9) Текст программы транслятора (в приложении, необязательно)

Задание

Транслятор должен запускаться из командной строки с несколькими входными параметрами. Первым и главным входным параметром должно быть имя файла программы на входном языке, вторым параметром может быть имя результирующего файла. Требования к остальным параметрам командной строки и управляющим ключам (если они необходимы) устанавливаются исполнителем самостоятельно. Командная строка должна быть достаточной для функционирования транслятора. Помимо интерфейса командной строки возможно наличие дополнительного интерактивного интерфейса пользователя у транслятора (в том числе и графического) по усмотрению исполнителя работы.

Описание входного языка разрабатывается исполнителем самостоятельно. Входная программа может быть разбита на строки произвольным образом, все пробелы и переводы строки должны игнорироваться транслятором. Текст входной программы может содержать комментарии любой длины, которые должны игнорироваться транслятором (вид комментария задается исполнителем).

В качестве выходного (результирующего) должен использоваться язык ассемблера процессоров типа Intel 80x86.

В случае если на вход транслятора подается входная программа, содержащая лексические, синтаксические или семантические ошибки, транслятор должен корректно завершать свое выполнение и выдавать сообщение о найденной ошибке во входной программе с указанием строки, в которой найдена ошибка. По возможности транслятор должен указывать тип найденной ошибки. Транслятор должен указать несколько ошибок во входной программе, если они были им обнаружены.

Рекомендации по выполнению задания

При выполнении работы надо обратить внимание на следующие основные моменты:

1. При работе над описанием разрабатываемого входного языка можно воспользоваться примером из [1], а именно 1-ой главой – при составлении учебного руководства по языку и приложением А – при написании справочника по языку. Учебное руководство по входному языку должно содержать примеры нескольких программ, иллюстрирующих основные свойства и область применения разрабатываемого языка. Справочник по языку должен содержать полное описание его лексической и синтаксической структуры, включая полную грамматику языка.
2. Построение грамматики входного языка – это определяющий момент в работе. Правильно построенная грамматика существенно упростит выполнение работы, а ошибки, напротив, могут существенно увеличить трудоемкость последующих этапов.

3. Создание лексического анализатора – это этап, не представляющий особой сложности. При выполнении этого этапа главная задача состоит в том, чтобы максимально эффективно разделить анализ, выполняемый лексическим анализатором, и анализ, выполняемый анализатором синтаксическим. Как правило, чем больше работы выполняет лексический анализатор, тем лучше. Уже построив грамматику языка, нужно иметь представление о том, какие элементы языка будут выделяться на этапе лексического анализа.
4. Выбор класса КС-грамматики для создания синтаксического анализатора – это второй по важности этап после построения грамматики. Важно построить грамматику входного языка так, чтобы она соответствовала методу синтаксического анализа, который планируется использовать, или же уметь преобразовать ее к требуемому виду.
5. Выбор формы внутреннего представления программы, методов оптимизации и генерации результирующего кода – это взаимозависимые процессы.

Необходимую дополнительную информацию можно найти в литературе по компиляторам и системам программирования.

Пример

Входным языком является язык со следующей грамматикой:
G ({0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, …, x, y, z, –, +, *, /, =, >, <, !, ;, {, }, (,)}, {F, S, E, L, I, V, B, D}, P, F)

P:
$F \rightarrow FS \mid \varepsilon$
$S \rightarrow ; \mid E; \mid$ print E; $\mid V = E; \mid$ while (E) S | if (E) S | if (E) S else S | {L}
$L \rightarrow S \mid LS$
$E \rightarrow I \mid V \mid –E \mid E + E \mid E – E \mid E * E \mid E / E \mid E < E \mid E > E \mid E >= E \mid E <= E \mid E != E \mid E == E \mid$ (E)
$I \rightarrow D \mid 1I \mid 2I \mid 3I \mid 4I \mid 5I \mid 6I \mid 7I \mid 8I \mid 9I$
$V \rightarrow a \mid b \mid c \mid d \mid e \mid \ldots \mid x \mid y \mid z$
$D \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

Общий заголовочный файл calc.h

Общий заголовочный файл содержит объявления для синтаксического дерева и таблицы символов (идентификаторов). Таблица символов (**sym**) состоит из односимвольных имен переменных. Узел синтаксического дерева может быть константой (**conNodeType**), идентификатором (**idNodeType**), или внутренним узлом с оператором (**oprNodeType**). Объединение включает в себя все три варианта, а **nodeType.type** используется для определения типа узла.

```
typedef enum { typeCon, typeId, typeOpr } nodeEnum;

/* константы */
typedef struct
{
     int value; /* значение константы */
} conNodeType;
```

```
/* идентификаторы */
typedef struct
{
     int i; /* индекс в таблице символов */
} idNodeType;

/* операторы */
typedef struct
{
     int oper; /* оператор */
     int nops; /* кол-во операндов */
     struct nodeTypeTag *op[1]; /* операнды */
} oprNodeType;

typedef struct nodeTypeTag
{
     nodeEnum type; /* тип вершины */
     /* объединение должно быть последним в nodeType, */
     /* так как operNodeType может динамически увеличиваться */
     union
     {
          conNodeType con; /* константы */
          idNodeType id; /* идентификаторы */
          oprNodeType opr; /* операторы */
     };
} nodeType;

extern int sym[26];
```

Входной файл для лексического анализатора

Входной файл для лексического анализатора содержит шаблоны для токенов **VARIABLE** и **INTEGER**. Кроме того, определяются токены для двухсимвольных операторов, таких как «==» или «!=». Для односимвольных операторов просто возвращается их значение.

```
%{
#include <stdlib.h>
#include "calc.h"
#include "y.tab.h"

void yyerror(char *);
%}

%%

[a-z] {
          yylval.sIndex = *yytext - 'a';
          return VARIABLE;
      }

0     {
```

```
          yylval.iValue = atoi(yytext);
          return INTEGER;
       }
[1-9][0-9]* {
               yylval.iValue = atoi(yytext);
               return INTEGER;
            }

[-()<>=+*/;{}] {
                    return *yytext;
                }

">=" return GE;
"<=" return LE;
"==" return EQ;
"!=" return NE;
"while" return WHILE;
"if" return IF;
"else" return ELSE;
"print" return PRINT;
[ \t\n]+ ; /* пробелы, табуляции и т.п. игнорируются */
. yyerror("Неизвестный символ");

%%

int yywrap(void)
{
     return 1;
}
```

Входной файл для синтаксического анализатора

Входной файл для YACC разделяется на три раздела. Раздел определений состоит из объявлений токенов и кода на Си в скобках “%{“ и “%}”. Описание грамматики помещается в раздел правил, а пользовательские процедуры в разделе процедур.

```
... определения ...
%%
... правила ...
%%
... процедуры ...
```

В YACC можно связать каждый символ грамматики, как терминальный, так и нетерминальный, с его значением. Например, если токен был объявлен как NUMBER, то его значением может быть конкретное число; если он был объявлен как STRING, значением может быть строка символов; если токен был объявлен как IDENTIFIER, его значением будет указатель на содержимое таблицы идентификаторов, описывающее этот идентификатор.

Каждый из этих видов значения соответствует своему типу данных в языке Си: int или double для NUMBER, char * для STRING и указатель на структуру для IDENTIFIER.
Объявление **%union** определяет все возможные типы языка Си, которые могут принимать терминальные и нетерминальные символы. YACC преобразует их в определение типа с именем YYSTYPE (typedef union YYSTYPE) в файле y.tab.h:

```
typedef union {
int iValue; /* целое число */
char sIndex; /* индекс в таблице символов */
nodeType *nPtr;
} YYSTYPE;
extern YYSTYPE yylval;
```

Теперь для каждого символа грамматики, чьё значение используется в синтаксическом анализе, нужно определить тип. Для нетерминалов это делается с помощью **%type**. А для терминальных символов с помощью **%token**, **%left**, **%right**, или **%nonassoc**.

Определение **%token <iValue> INTEGER** объявляет токен **INTEGER** и связывает его с **iValue** в объединении **YYSTYPE**.

А **%type <nPtr> stmt expr stmt_list** связывает **stmt, expr** и **stmt_list** с указателем **nPtr**.

Такие связывания необходимы для того, чтобы YACC мог сгенерировать правильный код. Например, для правила

expr: INTEGER { $$ = con($1); }

должен быть сгенерирован следующий код:

yylval.nPtr = con(yyvsp[0].iValue);

При этом **yyvsp[0]** – элемент верхушки стека, значение которого ассоциируется с **INTEGER**.

При использовании **$$** происходит обращение к значению левой части правила (слева от «:»). А **$1** означает обращение к первому значению на правой стороне правила, **$2** – ко второму значению и т.д.

Использование **$$**, **$1** и т.д. ведёт к автоматическому применению нужного поля объединения **%union**.

Далее следуют определения арифметических операторов и операторов сравнения. **%left** указывается для левоассоциативных, а **%right** – для правоассоциативных операций. Определение, описанное последним, имеет наивысший приоритет. Очевидно, что умножение и деление имеют более высокий приоритет, чем сложение и вычитание. В данном примере все операции являются левоассоциативными.

Унарный минус имеет более высокий приоритет, чем бинарные операторы, потому что его определение следует после них:

%left GE LE EQ NE '>' '<'
%left '+' '-'
%left '*' '/'
%nonassoc UMINUS

%nonassoc указывает на то, что ассоциативность в этом случае не используется. Далее в правиле, описывающем выражение с унарным минусом

expr: '-' expr %prec UMINUS { $$ = node(UMINUS, 1, $2); }

используется **%prec** для указания на то, что приоритет правила такой же, как и приоритет токена **UMINUS**.

Та же методика используется для удаления неоднозначности в операторе if-else. Пусть у нас есть следующие правила:

stmt:
IF expr stmt
| IF expr stmt ELSE stmt
...

и следующее состояние синтаксического анализатора ('.' показывает положение считывающей головки):

IF expr stmt IF expr stmt . ELSE stmt

Нам нужно решить, выполнять ли сдвиг **ELSE** или сворачивать **IF expr stmt** на верхушке стека. Если будет выполнен сдвиг:

IF expr stmt IF expr stmt . ELSE stmt

IF expr stmt IF expr stmt ELSE . stmt
IF expr stmt IF expr stmt ELSE stmt .
IF expr stmt stmt .

тогда второй **ELSE** – это пара для второго **IF**.

Если будет выполнена свертка

IF expr stmt IF expr stmt . ELSE stmt
IF expr stmt stmt . ELSE stmt
IF expr stmt . ELSE stmt
IF expr stmt ELSE . stmt
IF expr stmt ELSE stmt .

, тогда второй **ELSE** – это пара для первого **IF**.

В современных языках программирования **ELSE** связывается с самым последним **IF**. Следовательно, синтаксический анализатор должен идти по второму пути. Именно такое поведение и является поведением по умолчанию для YACC. Но при этом YACC инициирует предупреждающее сообщение о сдвиге-свертке. Для того чтобы это сообщение не выводилось, нужно задать для **IF-ELSE** более высокий приоритет, чем для простого **IF**:

%nonassoc IFX
%nonassoc ELSE
stmt:
IF expr stmt %prec IFX
| IF expr stmt ELSE stmt

Синтаксическое дерево строится снизу вверх. Листья дерева создаются при распознавании переменных и целых чисел. При нахождении оператора для него создается узел, а указатели на узлы, созданные на прошлом шаге, становятся операндами.

```
%{
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
#include "calc.h"

/* прототипы функций */
nodeType *opr(int oper, int nops, ...);
nodeType *id(int i);
nodeType *con(int value);
void freeNode(nodeType *p);
int ex(nodeType *p);
int yylex(void);
void yyerror(char *s);

int sym[26]; /* таблица символов */
%}

%union
{
     int iValue; /* целое число */
     char sIndex; /* индекс в таблице символов */
     nodeType *nPtr;
};

%token <iValue> INTEGER
%token <sIndex> VARIABLE
```

```
%token WHILE IF PRINT
%nonassoc IFX
%nonassoc ELSE
%left GE LE EQ NE '>' '<'
%left '+' '-'
%left '*' '/'
%nonassoc UMINUS
%type <nPtr> stmt expr stmt_list

%%

program:
     function { exit(0); }
     ;

function:
     function stmt { ex($2); freeNode($2); }
     | /* NULL */
     ;

stmt:
     ';'  {$$ = opr(';', 2, NULL, NULL);}
     | expr ';' {$$ = $1;}
     | PRINT expr ';' {$$ = opr(PRINT, 1, $2);}
     | VARIABLE '=' expr ';' {$$ = opr('=', 2, id($1), $3);}
     | WHILE '(' expr ')' stmt {$$ = opr(WHILE, 2, $3, $5);}
     | IF '(' expr ')' stmt %prec IFX {$$ = opr(IF, 2, $3, $5);}
     | IF '(' expr ')' stmt ELSE stmt
       {$$ = opr(IF, 3, $3, $5, $7);}
     | '{' stmt_list '}' {$$ = $2;}
     ;

stmt_list:
     stmt {$$ = $1;}
     | stmt_list stmt {$$ = opr(';', 2, $1, $2);}
     ;

expr:
     INTEGER { $$ = con($1); }
     | VARIABLE { $$ = id($1); }
     | '-' expr %prec UMINUS { $$ = opr(UMINUS, 1, $2); }
     | expr '+' expr { $$ = opr('+', 2, $1, $3); }
     | expr '-' expr { $$ = opr('-', 2, $1, $3); }
     | expr '*' expr { $$ = opr('*', 2, $1, $3); }
     | expr '/' expr { $$ = opr('/', 2, $1, $3); }
     | expr '<' expr { $$ = opr('<', 2, $1, $3); }
     | expr '>' expr { $$ = opr('>', 2, $1, $3); }
     | expr GE expr { $$ = opr(GE, 2, $1, $3); }
     | expr LE expr { $$ = opr(LE, 2, $1, $3); }
     | expr NE expr { $$ = opr(NE, 2, $1, $3); }
     | expr EQ expr { $$ = opr(EQ, 2, $1, $3); }
     | '(' expr ')' { $$ = $2; }
     ;
```

```
%%

#define SIZEOF_NODETYPE ((char *)&p->con - (char *)p)
nodeType *con(int value)
{
     nodeType *p;
     size_t nodeSize;
     /* Выделение памяти для вершины */
     nodeSize = SIZEOF_NODETYPE + sizeof(conNodeType);
     if ((p = malloc(nodeSize)) == NULL)
          yyerror("out of memory");
     /* Копирование данных */
     p->type = typeCon;
     p->con.value = value;
     return p;
}
nodeType *id(int i)
{
     nodeType *p;
     size_t nodeSize;
     /* Выделение памяти для вершины */
     nodeSize = SIZEOF_NODETYPE + sizeof(idNodeType);
     if ((p = malloc(nodeSize)) == NULL)
          yyerror("out of memory");
     /* Копирование данных */
     p->type = typeId;
     p->id.i = i;
     return p;
}

nodeType *opr(int oper, int nops, ...)
{
     va_list ap;
     nodeType *p;
     size_t nodeSize;
     int i;
     /* Выделение памяти для вершины */
     nodeSize = SIZEOF_NODETYPE + sizeof(oprNodeType) +
     (nops - 1) * sizeof(nodeType*);
     if ((p = malloc(nodeSize)) == NULL)
          yyerror("out of memory");
     /* Копирование данных */
     p->type = typeOpr;
     p->opr.oper = oper;
     p->opr.nops = nops;
     va_start(ap, nops);
     for (i = 0; i < nops; i++)
          p->opr.op[i] = va_arg(ap, nodeType*);
     va_end(ap);
     return p;
}
```

```
void freeNode(nodeType *p)
{
     int i;
     if (!p) return;
     if (p->type == typeOpr)
     {
          for (i = 0; i < p->opr.nops; i++)
               freeNode(p->opr.op[i]);
     }
     free (p);
}

void yyerror(char *s)
{
     fprintf(stdout, "%s\n", s);
}

int main(void)
{
     yyparse();
     return 0;
}
```

Компилятор

Функция **ex** вызывается для прохода по синтаксическому дереву в глубину. При этом осуществляется проход по узлам в том порядке, в каком они создавались. Это позволяет использовать операторы в порядке их нахождения при синтаксическом анализе.

```
#include <stdio.h>
#include "calc.h"
#include "y.tab.h"

static int lbl;
static int op;
static int flag;
static int con;

int ex(nodeType *p)
{
  int lbl1, lbl2;
  if (!p) return 0;
  switch(p->type)
  {
    case typeCon:
     switch(op)
     {
       case '=':
         con = p->con.value;
         break;
       case PRINT:
         printf("%d\n", p->con.value);
         break;
```

```
   default:
     printf(" %d\n", p->con.value);
     break;
 }
 break;
case typeId:
  switch(op)
  {
      case '=':
           printf("\tmov %c,%d\n", p->id.i + 'a', con);
           break;
      case PRINT:
           printf("%c\n", p->id.i + 'a');
           break;
      default:
           printf("%c,", p->id.i + 'a');
           break;
  }
  break;
  case typeOpr:
  op = p->opr.oper;
  switch(p->opr.oper)
  {
   case WHILE:
           printf("L%03d:\n", lbl1 = lbl++);
           ex(p->opr.op[0]);
           printf("\tjz\tL%03d\n", lbl2 = lbl++);
           ex(p->opr.op[1]);
           printf("\tjmp\tL%03d\n", lbl1);
           printf("L%03d:\n", lbl2);
           break;
   case PRINT:
           printf("\tprint ");
           ex(p->opr.op[0]);
           break;
   case '=':
           ex(p->opr.op[1]);
           if(!flag)
           {
             ex(p->opr.op[0]);
           }
           break;
   default:
           switch(p->opr.oper)
           {
                case '+':
                     flag = 1;
                     printf("\tadd ");
                     ex(p->opr.op[0]);
                     ex(p->opr.op[1]);
                     break;
                case '-':
                     flag = 1;
```

```
                    printf("\tsub ");
                    ex(p->opr.op[0]);
                    ex(p->opr.op[1]);
                    break;
                case '*':
                    flag = 1;
                    printf("\tmul ");
                    ex(p->opr.op[0]);
                    ex(p->opr.op[1]);
                    break;
                case '/':
                    flag = 1;
                    printf("\tdiv ");
                    ex(p->opr.op[0]);
                    ex(p->opr.op[1]);
                    break;
                case '<':
                case '>':
                case GE:
                case LE:
                case NE:
                case EQ:
                    printf("\tcmp ");
                    flag = 1;
                    ex(p->opr.op[0]);
                    ex(p->opr.op[1]);
                    break;
                default:
                  ex(p->opr.op[0]);
                  ex(p->opr.op[1]);
                  break;
            }
        }
  }
  return 0;
}
```

Результат работы компилятора

Исходный текст программы:

```
x = 0;
while (x < 3)
{
   print x;
   x = x + 1;
}
```

Результат работы компилятора:

```
    mov x,0
L000:
    cmp x, 3
```

```
    jz      L001
    print  x
    add x, 1
    jmp     L000
L001:
```

Литература

1. Керниган Б., Ричи Д. Язык программирования Си. – 2-е изд. – М.: Вильямс, 2009. – 304 с.
2. Молчанов А.Ю. Системное программное обеспечение: Учебник для вузов. 3-е изд. – СПб.: Питер, 2010 год. – 400 с.
3. Ахо А.В., Лам М. С., Сети Р., Ульман Дж. Д. Компиляторы: принципы, технологии и инструментарий, 2-е изд.: Пер. с англ. – М.: ООО Изд. дом Вильямс, 2008. – 1184 с.
4. Fast Lexical Analyzer. Режим доступа: http://flex.sourceforge.net/
5. Levine J.R. Flex & bison. – O'Reilly Media, Inc., 2009. – 274 p.
6. Niemann T. A Compact Guide To Lex & Yacc. Режим доступа: http://epaperpress.com/lexandyacc/download/lexyacc.pdf
7. Серебряков В.А., Галочкин М.П. Основы конструирования компиляторов. – М.: Едиториал УРСС, 2001. – 224 с. (http://www.citforum.ru/programming/theory/serebryakov/)
8. Свердлов С.З. Языки программирования и методы трансляции: Учебное пособие. — СПб.: Питер, 2007. — 638 с.
9. Голубь Н.Г. Искусство программирования на Ассемблере. Лекции и упражнения. – М.: DiaSoft, 2002. – 656 с.
10. Рудаков П.И., Финогенов К.Г. Язык ассемблера: уроки программирования. – М.: ДИАЛОГ-МИФИ, 2001. – 640 с.